Chapter 5
CPU Scheduling

Da-Wei Chang

CSIE.NCKU

Source: Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, "Operating System Concepts", 10th Edition, Wiley.

Outline

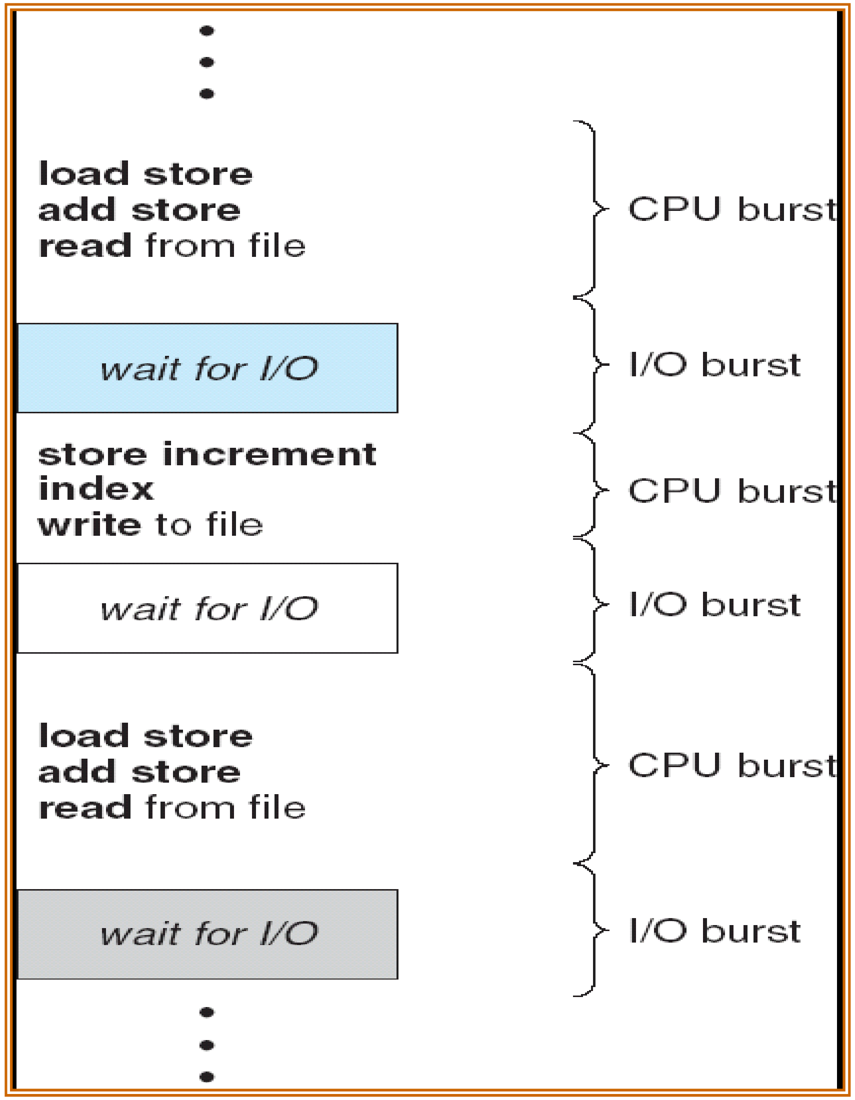
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Thread Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Basic Concepts

- Scheduling is a basis of multiprogramming
  - Switching the CPU among processes improves CPU utilization

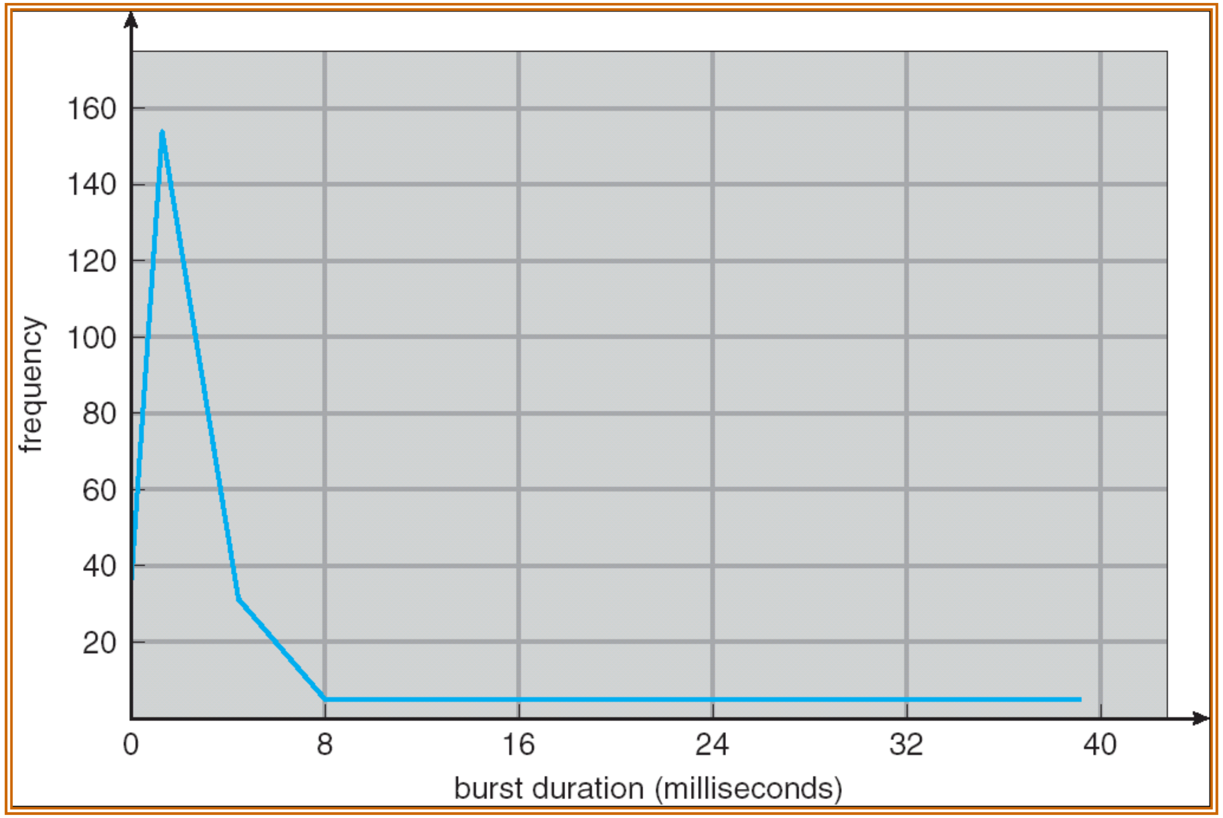
- CPU-I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait

Alternating Sequence of CPU and I/O Bursts



Histogram of CPU-burst Times

CPU Burst Distribution



A large # of short CPU bursts and a small # of long CPU bursts

IO bound → many short CPU bursts, few long CPU bursts
CPU bound→ more long CPU bursts

CPU Scheduler

- Short term scheduler
- Selects among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
    1. Switches from running to waiting state (IO, wait for child)
    2. Switches from running to ready state (timer expire)
    3. Switches from waiting to ready (IO completion)
    4. Terminates

Non-preemptive vs. Preemptive Scheduling

- **Non-preemptive Scheduling/Cooperative Scheduling**
  - Scheduling takes place only under circumstances 1 and 4
  - Process holds the CPU until termination or waiting for IO
  - MS Windows 3.1; Mac OS (before Mac OS X)
  - Does not require specific HW support for preemptive scheduling
    - E.g., timer
- **Preemptive Scheduling**
  - Scheduling takes place under all the circumstances (1 to 4)
  - Better for time-sharing system and real-time systems
  - Usually, more context switches
  - A cost associated with shared data access
    - May be preempted in an unsafe point

Scheduling Criteria

- Used to judge the performance of a scheduling algorithm

- **CPU utilization**
  – (100% - ratio of CPU idle)
- **Throughput**
  – # of processes that complete their execution per time unit
- **Turnaround time**
  – amount of time to execute a particular process
  – From process submission to process termination

Scheduling Criteria

- **Waiting time**
  - amount of time a process has been waiting **in the ready queue**
  - Scheduler does not affect the time for
    - Execution instructions
    - Performing IOs

  Here, we do not consider the memory (including cache) effect

- **Response time**
  - amount of time it takes from when a request was submitted until the **first** response is produced, **not** output (for time-sharing environment)

Optimization Criteria

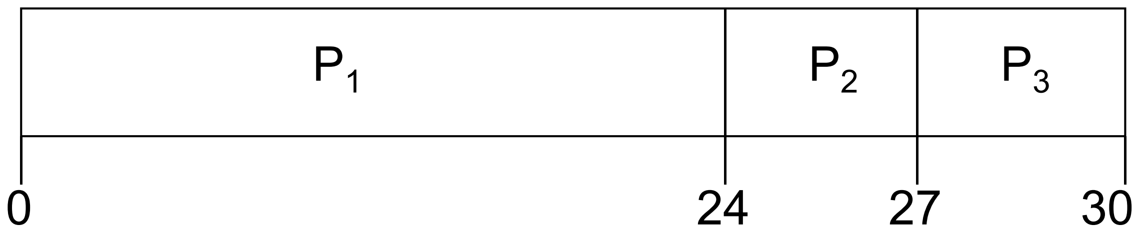
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

← **CPU burst**

- Implemented via a FIFO queue
- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



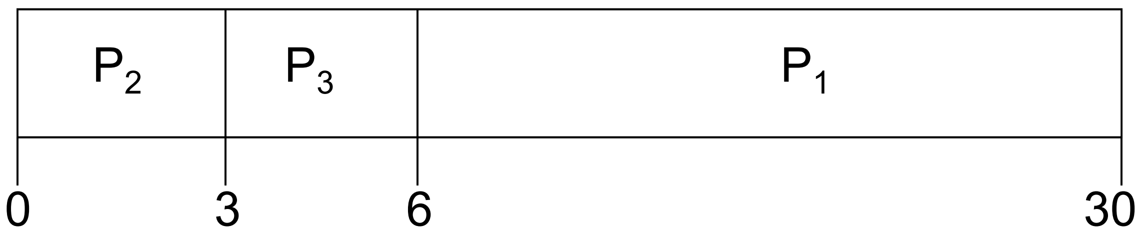
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case

FCFS Scheduling (Cont.)

- *Convoy effect*
  - Short process behind long process
  - Multiple IO bound process may wait for a single CPU bound process
    - Device idle….

- FCFS is non-preemptive
  - Not good for time-sharing systems

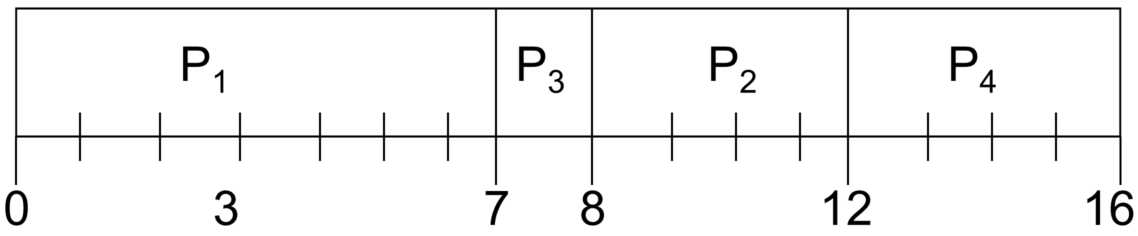
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst, and select the process with the shortest burst to run
- Two schemes:
  - nonpreemptive – once the CPU is given to a process, it cannot be preempted until the completion of the CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt the current process.
    - known as the Shortest-Remaining-Time-First (SRTF) scheduling
- SJF is optimal – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

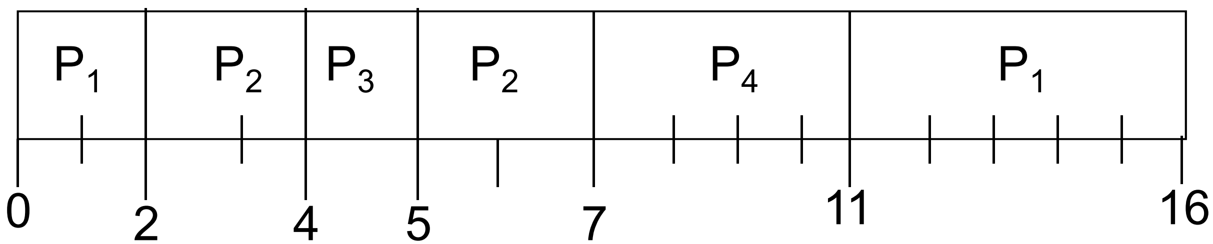


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

SJF Scheduling

- How to know the length of the next CPU burst?
  - Difficult…..
  - There is no easy way to know the length of the next CPU burst
  - So, ~~guess~~ predict it ….

Predicting Length of Next CPU Burst

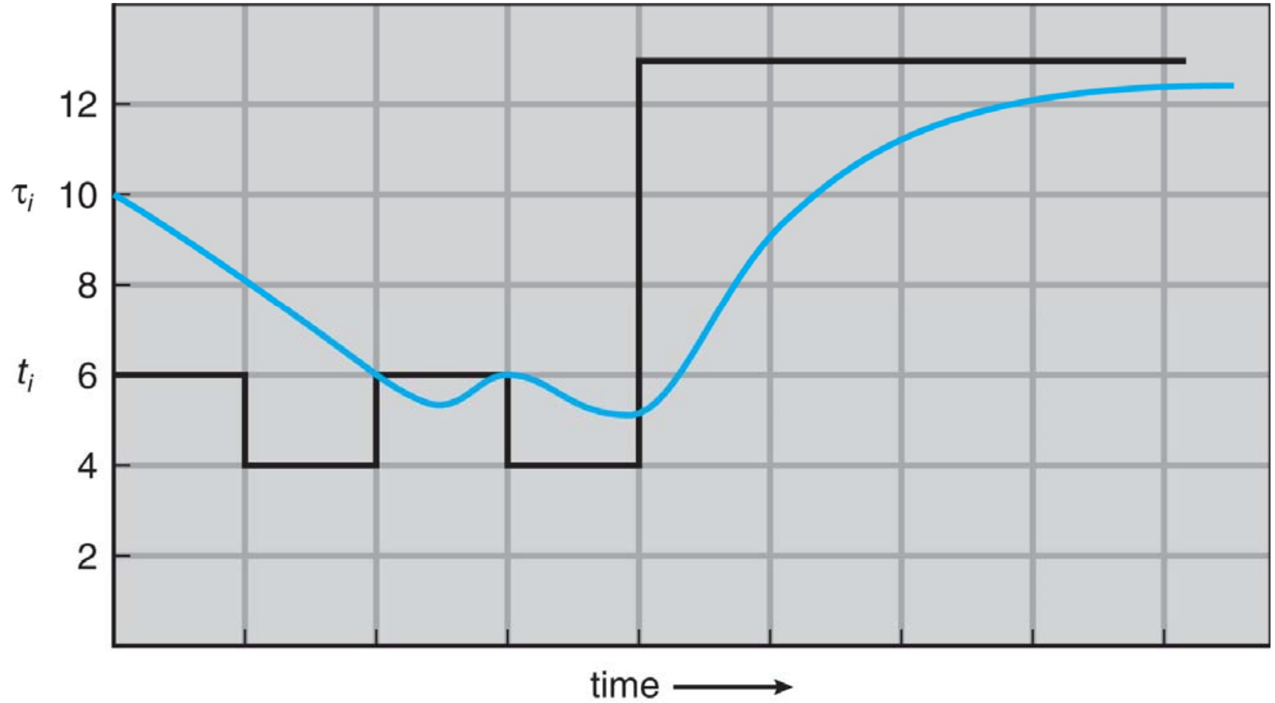
- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. $t_n =$ actual lengh of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

Predicting Length of the Next CPU Burst

$\tau_0 = 10$
$\alpha = 1/2$



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$
\begin{aligned}
\tau_{n+1} = {} & \alpha\, t_n + (1-\alpha)\alpha\, t_{n-1} + \ldots \\
& + (1-\alpha)^j \alpha\, t_{n-j} + \ldots \\
& + (1-\alpha)^{n+1} \tau_0
\end{aligned}
$$

- Since both α and $(1-\alpha)$ are typically less than 1, each successive term has less weight than its predecessor

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- A process will leave the running state if
  - Time quantum expire
  - Wait IO or events
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- RR is preemptive

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

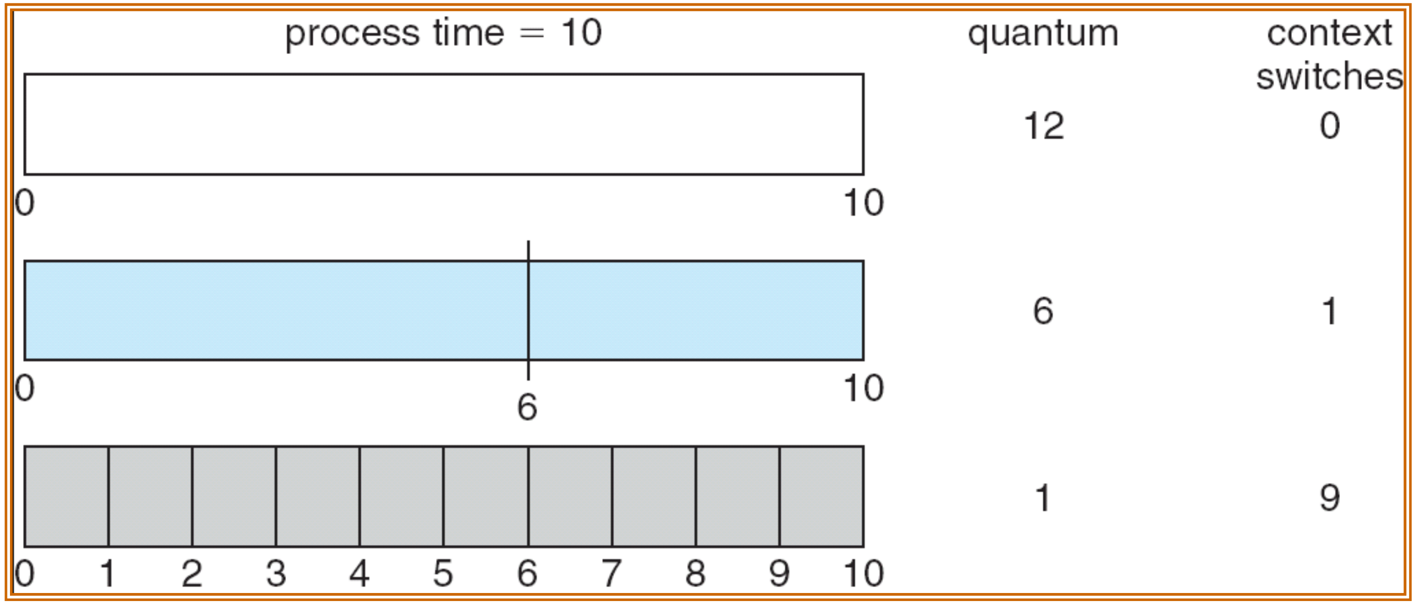
0 20 37 57 77 97 117 121 134 154 162

- Typically, longer average turnaround time than SJF, but better *response* time

Time Quantum and Context Switch Counts

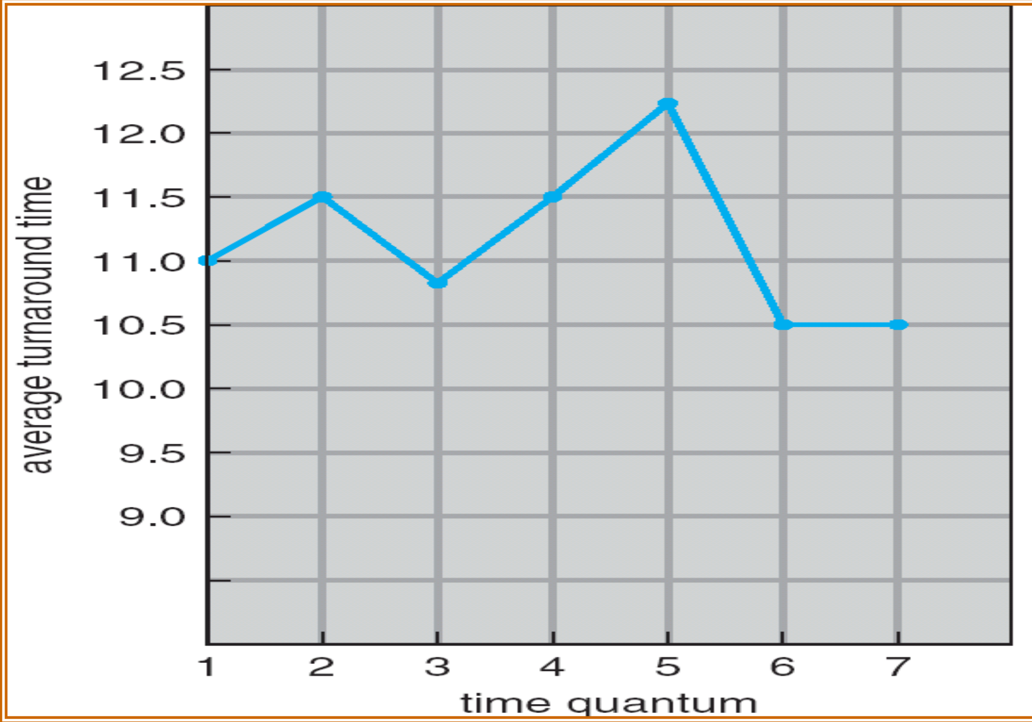
Performance

- q large $\Rightarrow$ FIFO
- q small $\Rightarrow$ a large number of context switches
  - q must be **large** with respect to context switch time, otherwise overhead is too high



Context switches are not free!!!

Turnaround Time Varies with the Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Given 3 processes of 10 time units

for quantum of 1 time unit ➔ average turnaround time = 29

for quantum of 10 time unit ➔ average turnaround time = 20

Rule of thumb: 80% of the CPU bursts should be shorter than the time quantum

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (in many systems, smallest integer ➔ highest priority)
  - Preemptive
  - Non-preemptive

Process	**Burst Time**	**Priority**
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Execution Sequence: P2, P5, P1, P3, P4

Priority Scheduling

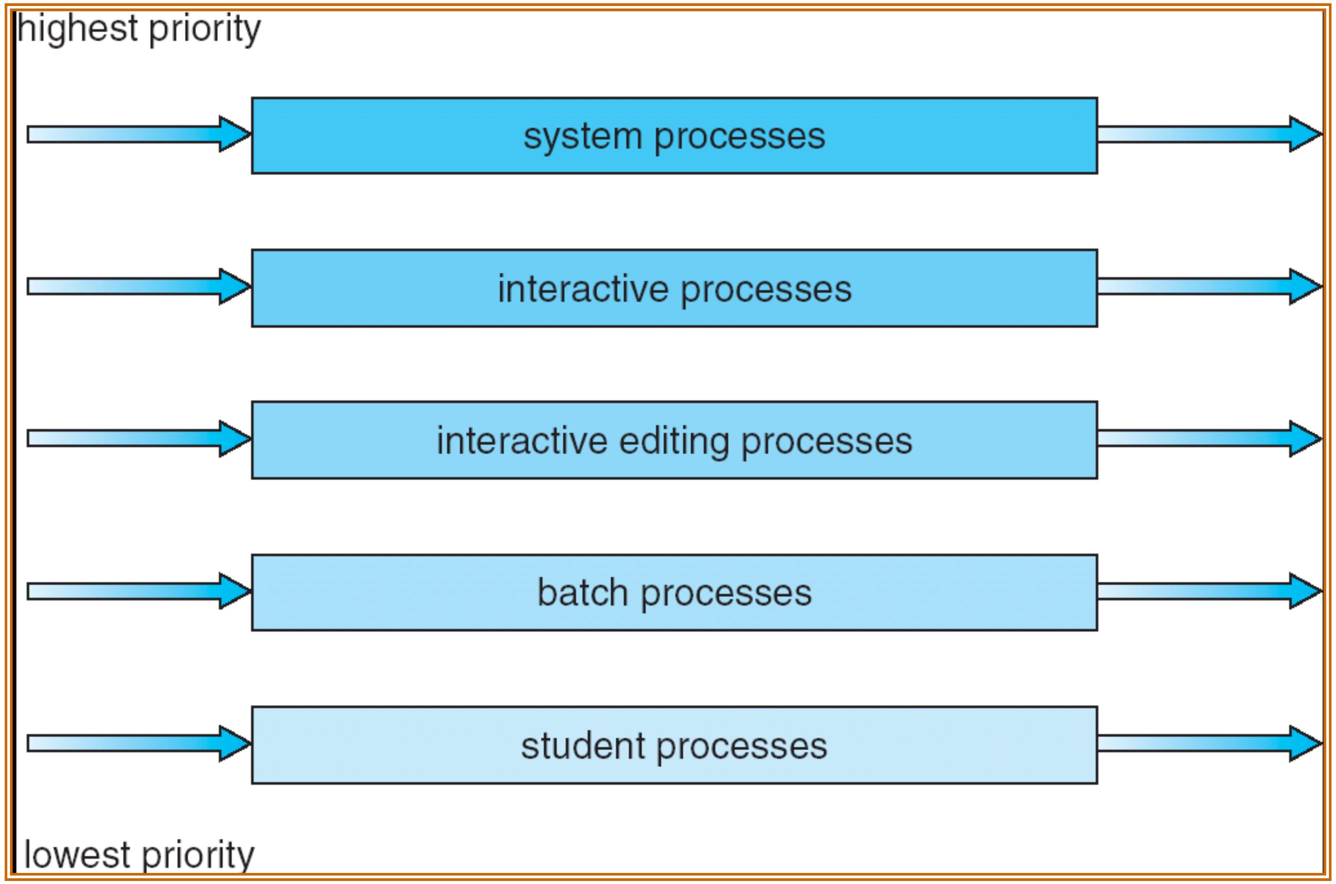
- The concept is general
  - SJF is a priority scheduling where priority is set according to the predicted next CPU burst time

- Problem : Starvation – low priority processes may never execute
  - A low priority process submitted in 1967 had not been run when the system IBM 7094 at MIT was shutdown in 1973
- Solution : Aging – as time progresses increase the priority of the process

Multilevel Queue

- Used when processes are easily classified into different groups
- Ready queue is partitioned into separate queues
  - e.g., foreground (interactive) and background (batch)
    - These two types of processes have different response time requirements
    - FG processes can have priority over BG processes
- A process is fixed on one queue
- Each queue has its own scheduling algorithm
  - E.g., foreground – RR; background – FCFS

Multilevel Queue Scheduling



Multilevel Queue

- Scheduling must be done between the queues
  - Fixed priority scheduling
    - i.e., serve all from foreground then from background
    - possibility of starvation
  - Time slice
    - each queue gets a certain amount of CPU time which it can schedule amongst its processes
    - i.e., 80% to foreground in RR, 20% to background in FCFS

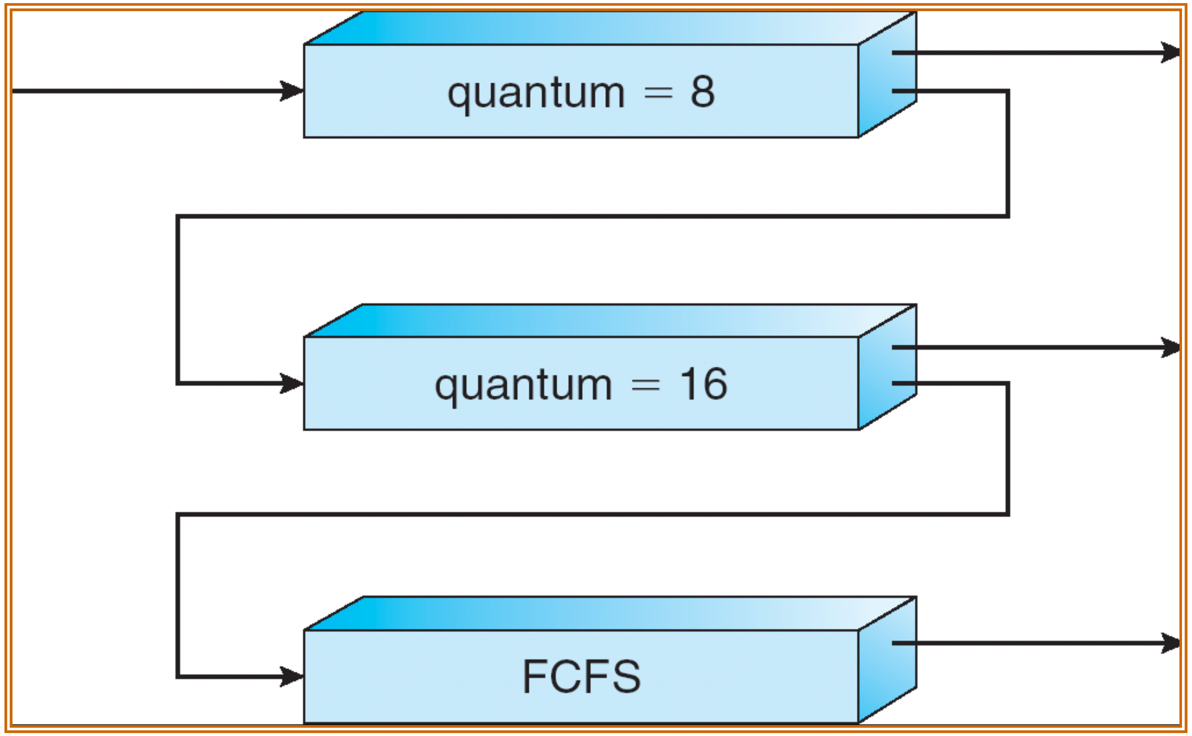
Multilevel Feedback Queue

- A process can move among different queues
- The idea
  - Separate processes according to the characteristics of their CPU bursts
    - Use too much CPU time ➔ move to a lower priority Q
      - Favor interactive and IO bound processes
    - Wait too long in a low priority Q ➔ move to a higher priority Q
      - aging

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0. When it gains CPU, job receives 8 milliseconds. If it does not finish its current burst in 8 milliseconds, job is preempted and moved to queue Q_1.
  - At Q_1 job is again served and receives 16 additional milliseconds. If it still does not complete its burst, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Give highest priority to processes with CPU burst <= 8ms

Multilevel Feedback Queues

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service
- It is the most generic algorithm
  - Can be configured to match a specific system
- It is the most complex algorithm
  - You have to select a proper value for each parameter

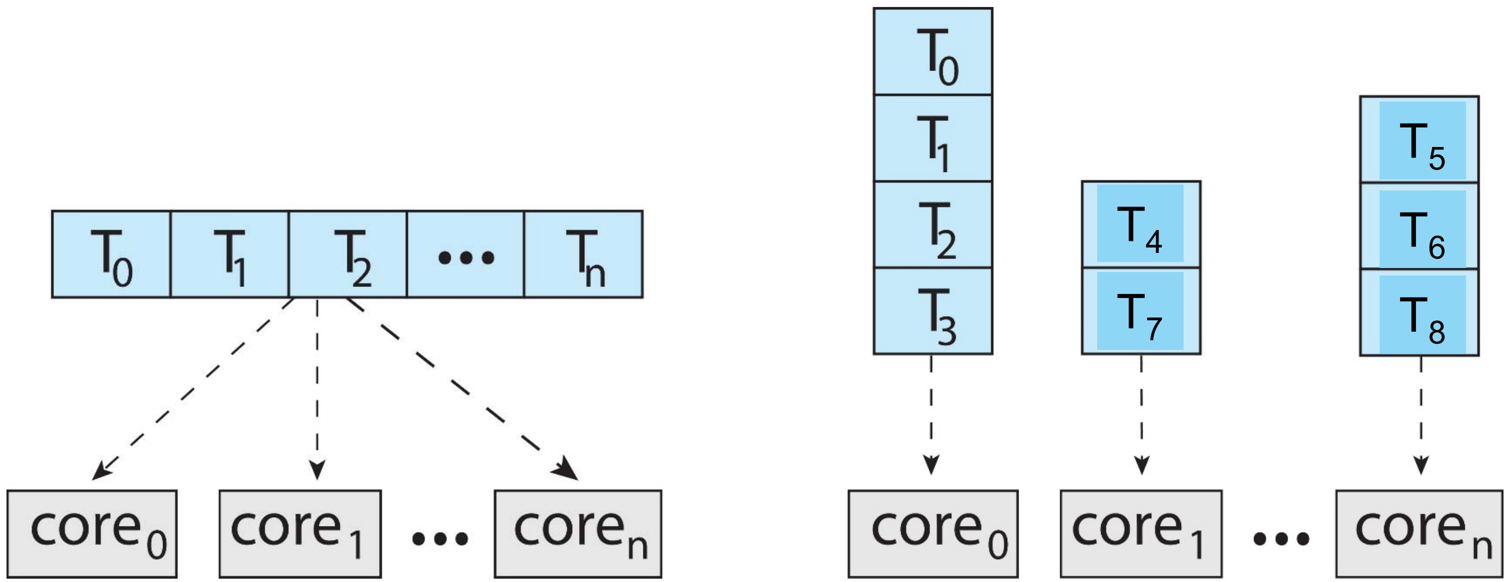
Multiple-Processor Scheduling

- Load sharing
- CPU scheduling is more complex when multiple CPUs are available
- We consider homogeneous processors
  - Can use any available processor to run any ready processes
- Topics
  - ASMP vs. SMP
  - Processor affinity
  - Load balancing
  - Multithreaded core

ASMP vs. SMP

- Approaches to MP scheduling
  - **Asymmetric multiprocessing (ASMP)**
    - Only one processor accesses the OS data structures, alleviating the need for data sharing
    - The other processors run user code only
  - **Symmetric multiprocessing (SMP)**
    - **All** processors can access the OS data structures
    - Each processor is self scheduling
    - Common or private ready queue *(see next slide)*
    - Scheduler in each processor selects a process from the ready Q
      - In case of common ready Q, must ensure
        - » Two processors don't choose the same process
        - » Processes are not lost from the Q
    - All modern OSs supports SMP
      - Win 2000, XP, Linux, Solaris, Mac OS X...
- *We focus on SMP systems here*

Common/Private Ready Queues



Common ready Q **Private ready Qs**

Processor Affinity

- Cache miss rate increases if a process migrates to another processor
- Most SMP systems try to avoid migration
  - Processor affinity
    - Keep the process running on the same processor
- Soft affinity
  - Try to keep the process always on a fixed processor
  - But, NO guarantee…
- Hard affinity
  - Guarantee to keep a process always on a fixed processor
  - Linux provides system calls to support hard affinity

Load Balancing

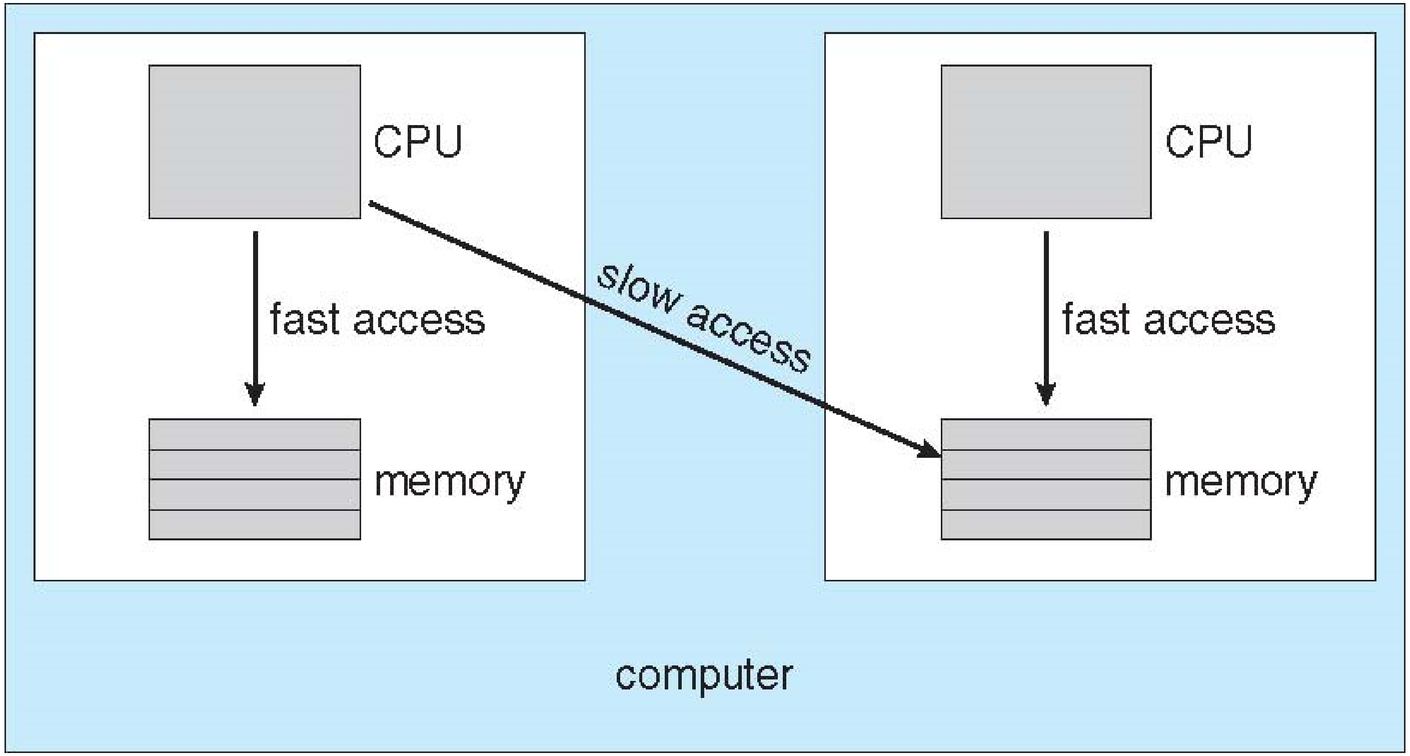
- Balance the load among the processors
- Only necessary on private-ready-Q systems
  - Different ready Qs can have different lengths
  - In common-ready-Q systems, the load is already balanced
  - Most contemporary OSs use private ready Qs
- Two general approaches
  - Push migration
    - a specific task periodically checks the load and balance the load if it finds an imbalance
  - Pull migration
    - An idle processor pulls a ready task from a busy processor
- The above two approaches can co-exist
  - Linux supports both (Note: It performs push migration every 200ms)

Load Balancing

- Load balancing often counteract the benefits of processor affinity
  - Load balancing is done by process migration
  - Processor affinity try not to migrate processes
  - An idle processor can
    - Pull processes only when imbalance exceeds a certain threshold

NUMA and CPU Scheduling

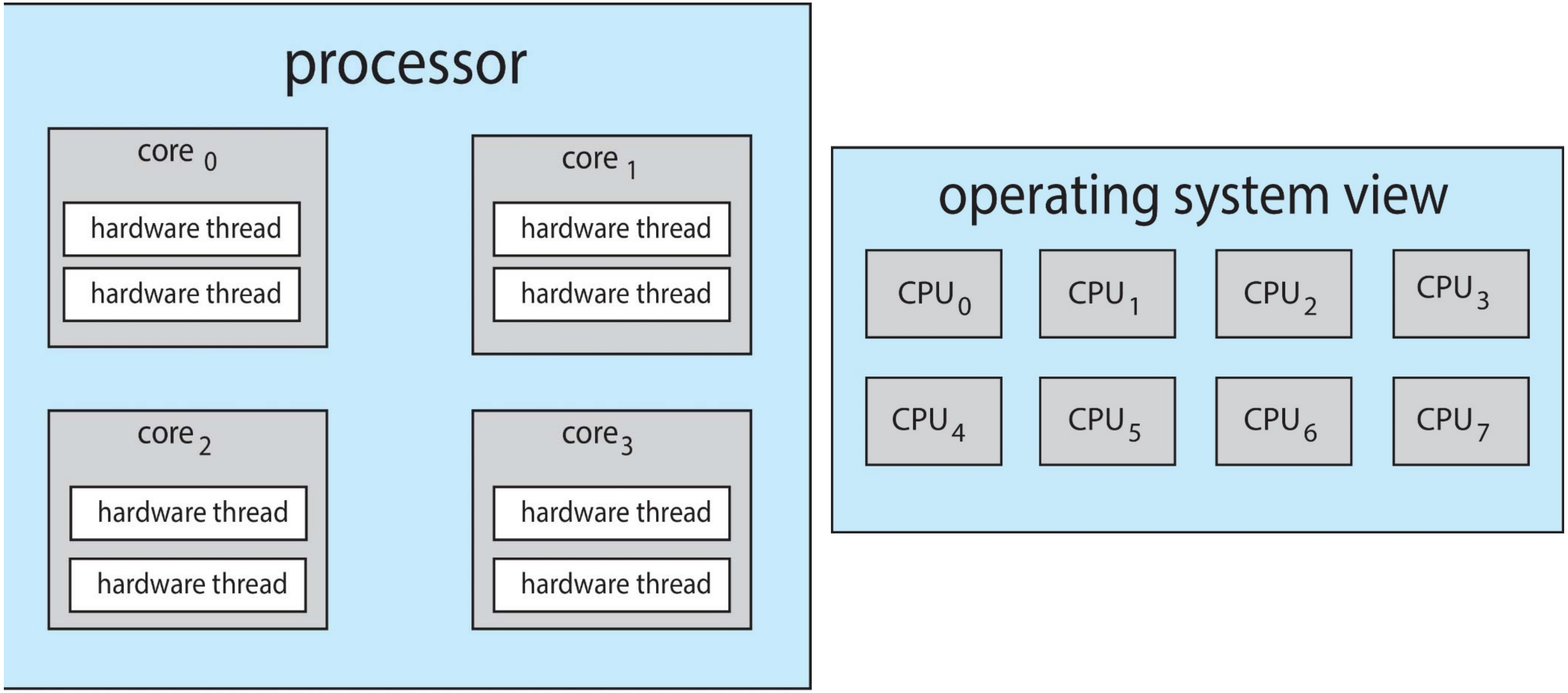
NUMA architecture also has affinity and load balancing issues…



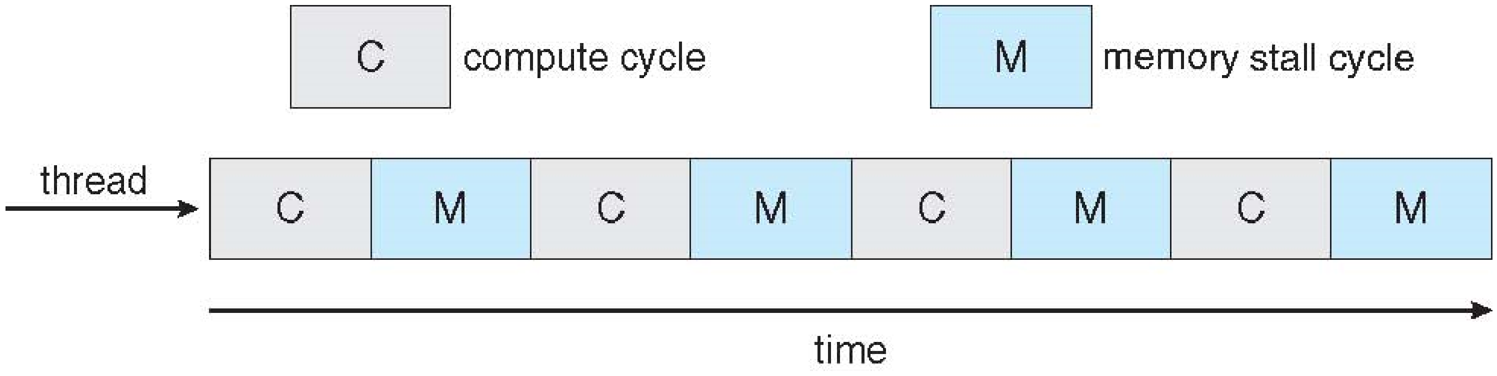
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
  - Faster and consume less power
- Multiple (hardware) threads per core also growing
  - Provide multiple **logical** (not physical) processors on the same physical core *(see next slide)*
    - Each logical P has its own architecture state
      - General and status registers
    - Each logical P handle its own interrupts
    - Logical Ps share the resources of the physical P such as ALU, cache, FPU..
    - E.g. Intel's hypertheading technology
  - **Idea:** Takes advantage of memory stall to make progress on another thread while memory retrieve happens

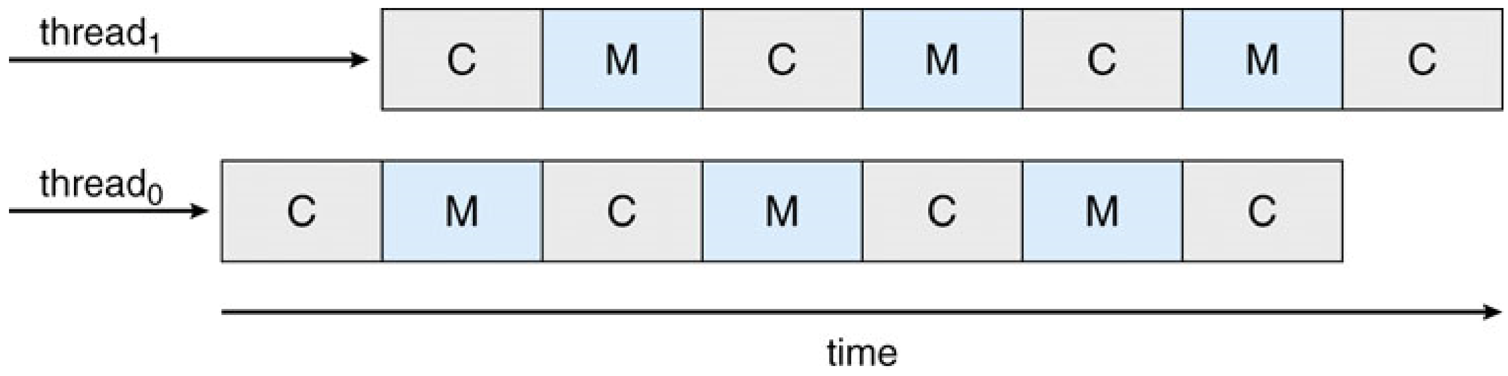
A Multithreaded Multicore System



Multithreaded Multicore System



A single-threaded core



A dual-threaded core

Real-Time Scheduling

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time

- *Soft real-time* computing – requires that critical processes receive priority over the others; NO guarantee on the execution time limit

Thread Scheduling

- Kernel threads are scheduled by OS
- User threads are managed by thread library

- Local Scheduling – How the threads library decides which thread to put onto an available LWP (kernel thread)
  - For M:1 or M:M models

- Global Scheduling – How the kernel decides which kernel thread to run next

Contention Scope

- Process Contention Scope (PCS)
  - Competitions among threads of the same process
  - Scheduling is typically done according to priority
    - Thread priorities are set by programmers, not adjusted by thread lib
    - Usually no time slicing among threads of equal priority
- System Contention Scope (SCS)
  - Competitions among threads in the system
  - Systems with 1:1 model only use SCS
    - Linux, MacOS…

Pthread Scheduling

- Contention Scope
  - PTHREAD_SCOPE_SYSTEM
  - PTHREAD_SCOPE_PROCESS
- On M:M systems
  - PTHREAD_SCOPE_PROCESS schedules the user thread onto available (and shared) LWPs
    - # of LWPs is determined by the thread lib
  - PTHREAD_SCOPE_SYSTEM will bind the user thread to a dedicated LWP
    - Becomes 1:1
- API
  - pthread_attr_setscope()
  - pthread_attr_getscope()

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
     int i;
    pthread_t tid[NUM THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RR, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
    /* create the threads */
    for (i = 0; i < NUM THREADS; i++)
            pthread_create(&tid[i], &attr, runner, NULL);
```

Pthread Scheduling API

```
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
            pthread_join(tid[i], NULL);
} /* end of main() */

/* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread_exit(0);
}
```

Operating System Examples

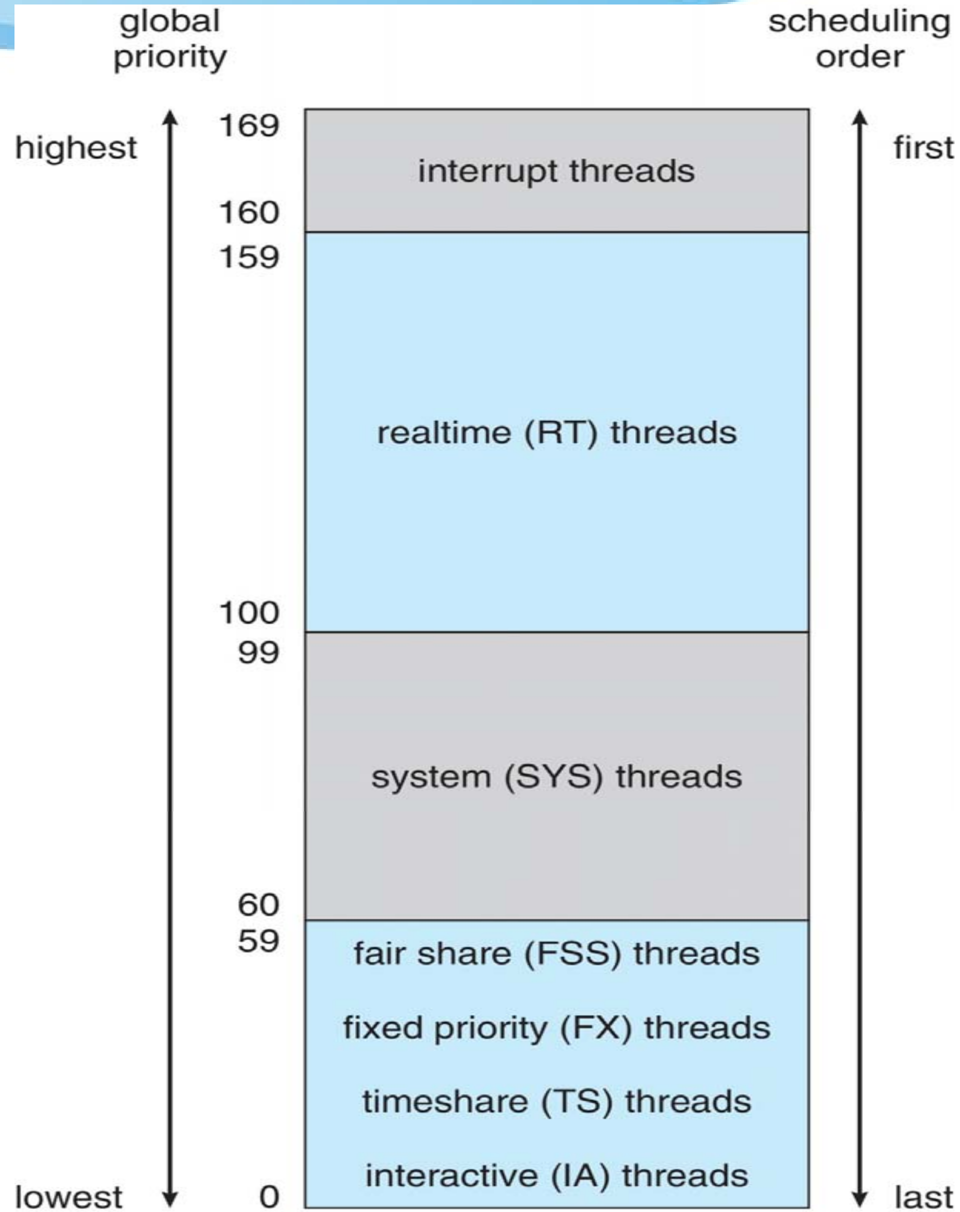
- We mention kernel thread scheduling here

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Solaris Scheduling

- Priority based
  - RR for same-priority threads
- 6 classes
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
  - Time sharing (TS) -- default
  - Interactive (IA)

Solaris Scheduling



Solaris Scheduling

- Real time (RT) class
  - The highest priority among the 6 classes
  - Allows a RT process to have fast responses
- System class
  - Kernel processes, such as paging daemon
- TS/IA classes
  - Dynamically alters priorities
  - Assign time slices of different lengths using a multilevel feedback Q
    - Higher priority ➔ smaller time slice
      - Good response time for interactive processes
      - Good throughput for CPU-bound processes

Solaris Dispatch Table for TS/IA Classes

Lowest priority

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Priority change to

favor IO bound processes

Windows XP Scheduling

priority-based preemptive scheduling

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Priority classes

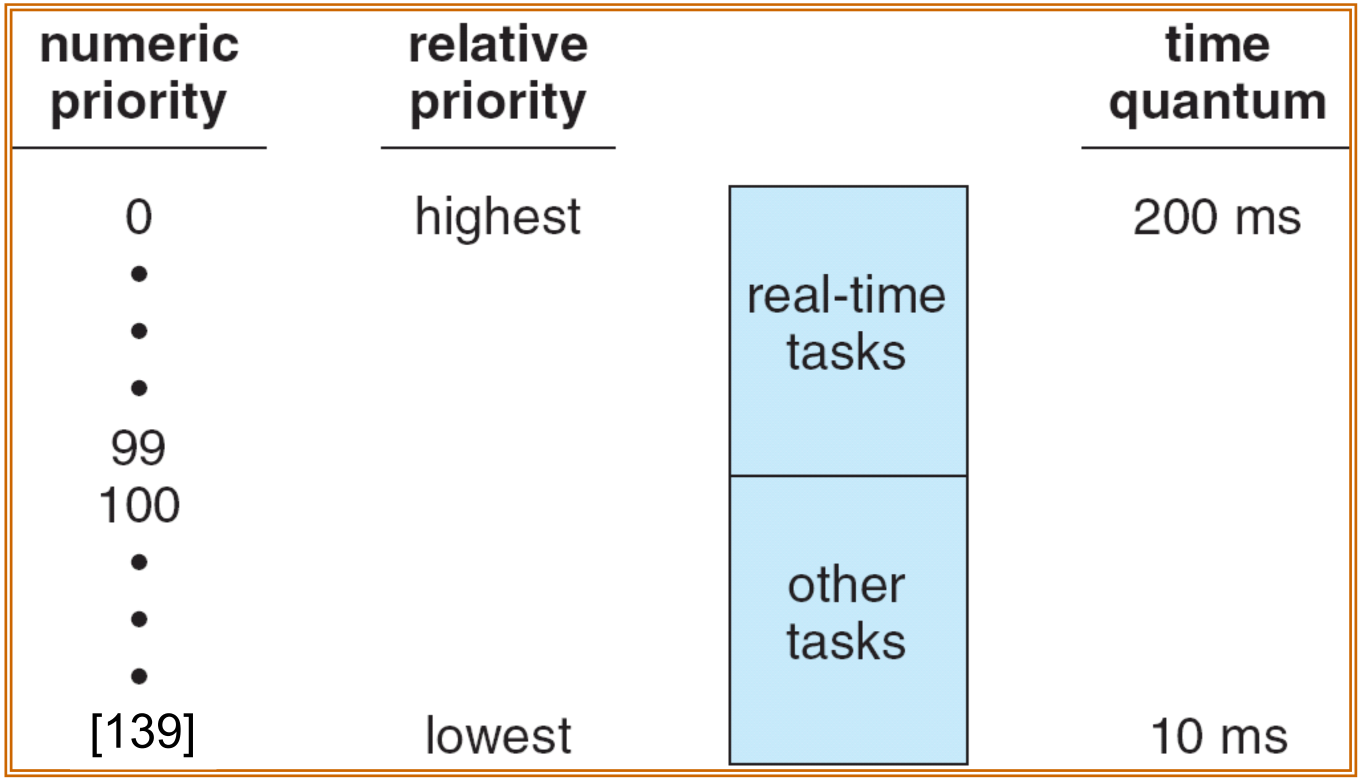
Relative Priority in a class

Increase the quantum of the foreground process by some factor (e.g., 3)

Linux Scheduling

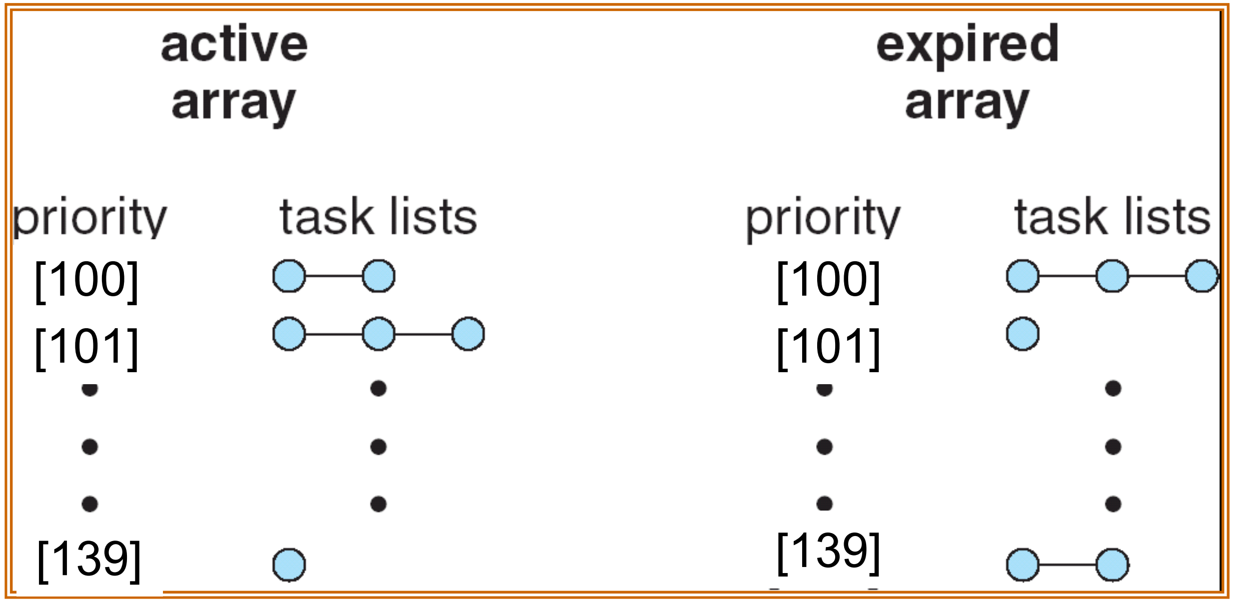
- Two algorithms: time-sharing and real-time (soft)
- Time-sharing
  - O(1) scheduler (kernel 2.5)
    - Prioritized & credit-based
    - Priority boosts for interactive or IO bound processes
    - Credit subtracted when timer interrupt occurs
    - When credit = 0, another process chosen
    - When all processes have credit = 0, re-crediting occurs
      - Based on factors including priority and history
  - CFS (after kernel 2.6)
- Real-time
  - Soft real-time
  - POSIX.1b compliant (IEEE 1003.1b-1993) – two classes
    - FCFS and RR
    - Highest priority process always runs first

O(1) Scheduler - Relationship between Priorities and Time-slice Length



O(1) Scheduler - List of Tasks Indexed According to Priorities

Each ready Q contains two arrays



Linux CFS Scheduler (after kernel 2.6)

- Completely Fair Scheduler (CFS)
- Schedule task with the smallest score
  - Derived from virtual run time of the task
  - Tasks with the smallest virtual run time tend to be selected to run
- nice value can affect the score
  - nice > 0 (lower priority) → increase score
  - nice < 0 (higher priority) → decrease score

Java Thread Scheduling

- JVM uses a Preemptive, Priority-based scheduling algorithm

- FIFO queue is used if there are multiple threads with the same priority

Java Thread Scheduling (cont.)

JVM Schedules a Thread to Run When:

1. The currently running thread exits the Runnable state
2. A higher priority thread enters the Runnable state

JVM doesn’t ensure time-slicing

Time-Slicing

Since the JVM doesn't ensure time-slicing, the yield() method may be used:

```
while (true) {
        // perform CPU-intensive task
        . . .
        Thread.yield();
}
```

This yields control to another thread of equal priority

Thread Priorities

Priority	Comment
Thread.MIN_PRIORITY	Minimum Thread Priority
Thread.MAX_PRIORITY	Maximum Thread Priority
Thread.NORM_PRIORITY	Default Thread Priority

Priorities may be set by using the setPriority() method:

setPriority(Thread.NORM_PRIORITY + 2);

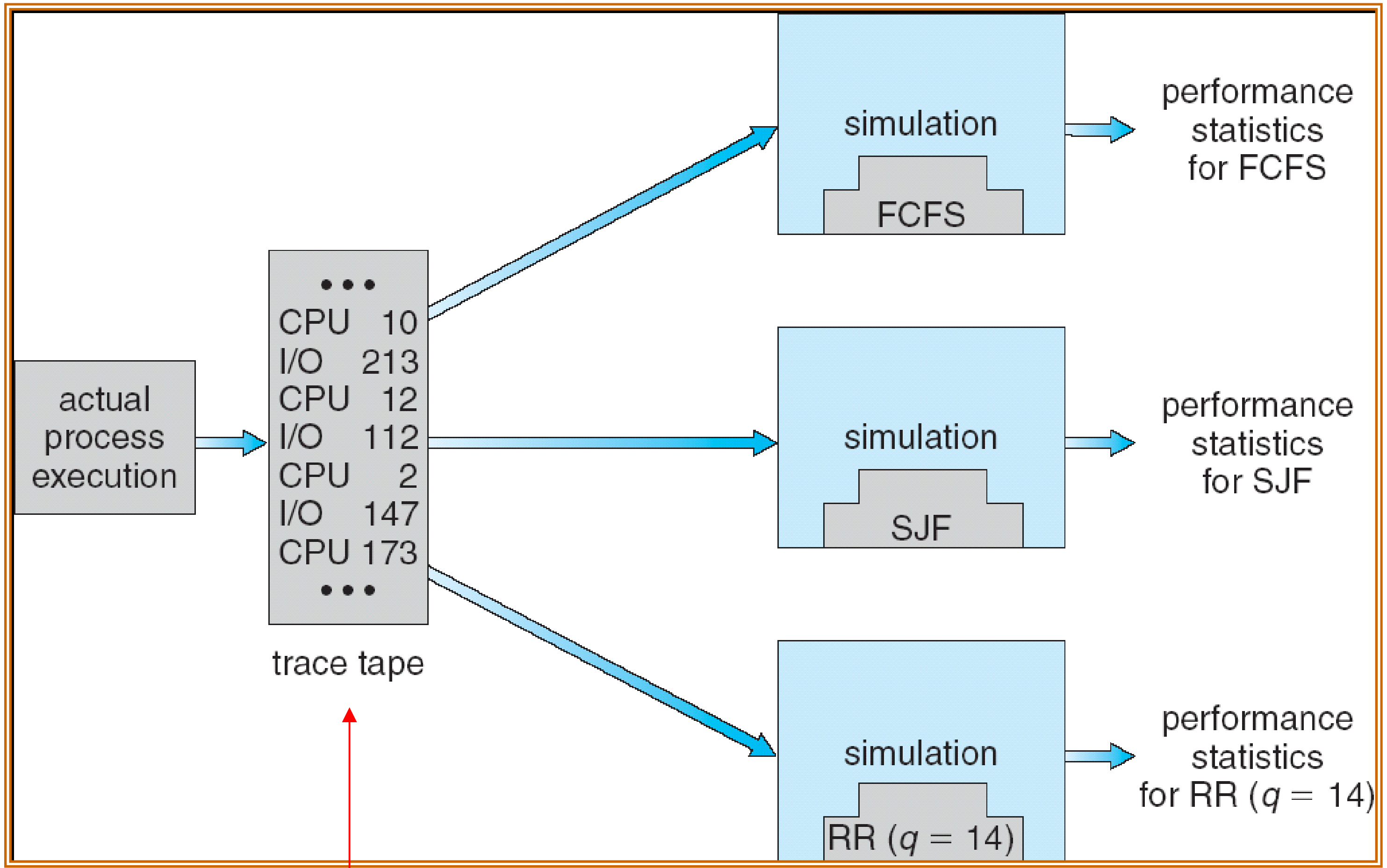
Algorithm Evaluation

- Deterministic modeling
  - takes a particular predetermined workload and defines the performance of each algorithm for that workload
  - Simple and fast
    - Useful only when the set of programs and their behaviors are fixed
- Queueing models
  - Assumes the distribution of the burst length and process arrival rates
  - It's possible to compute the average throughput, utilization, waiting time…

Algorithm Evaluation

- Simulation
  - *see next slide*
  - Still of limited accuracy
- Implementation
  - The only completely accurate way to evaluate an algorithm
  - High cost

Simulation



Traces of actual events